CS 61BL Final

Summer 2021 — August 12, 2021

INSTRUCTIONS

- This test has 11 questions, and is to be completed in 172 minutes. Note this exam was originally delivered online on PrairieLearn, and the extra two minutes were meant for saving work.
- The exam is closed book, except that you are allowed to use unlimited written notes.
- No calculators or other electronic devices are permitted.
- The exam is out of **54 points**. Note this exam was mistakenly given out of 53 points, so one free point was added. We wish you all the best of luck!

POINTS:

#	Points
1	0
2	3.5
3	6
4	5
5	6
6	5.5
7	4.25
8	5.5
9	5
10	6.25
11	6
Free Point	1
TOTAL:	54

1 Summer 2021 Final Welcome

Many of the problems on this exam consist of multiple choice or fill in the blank questions. Make sure to read directions carefully! Some questions may ask you to select one answer:

◯ you can only
◯ pick one of these

While others may ask you to select all answers that apply:

☐ you can pick ☐ both of these
For fill in the blank questions, your final answer should be a number unless otherwise specified. Do not include units or math expressions in your answer. For example, a valid answer would be "20" (without the quotes), but "20 nodes" or "5 * 4" would be invalid.

If you don't select an option for every multiple choice question and try to save your work, it will say "invalid, not gradable". Note this is expected, and your answer was saved.

There are also a few code writing questions. If you are given skeleton code to fill in, **we will not grade your answer if you alter the skeleton code** unless explicitly permitted. If you are given a line limit, **we will not grade your answer if you go over the line limit or if you don't properly format your code in the attempt of using fewer lines.**For instance, the properly formatted code below takes 9 lines, counting the function signature and closing brace.

```java
public static boolean function() {
    for (int i = 0; i < 10; i += 2) {
        if (i == 2) {
            i -= 1;
        } else {
            i += 1;
        }
    }
}
```

However, the code below is **not**, as it is using improper formatting in an attempt to evade the line limit!

```java
public static boolean function() {
    for (int i = 0; i < 10; i += 2) {
        if (i == 2) { i -= 1; }
        else { i += 1; } }
}
```

Additionally, **we will not grade your question if you fail to follow any restrictions given in the problem statement**. Like in previous exams, your code won't be checked by a compiler, but we will take off points for errors that are more than a typo. Please pay attention to detail when answering coding questions!

Please check the following boxes to confirm that you understand the exam proctoring policy!

☐ I understand that I may not use any internet resources or communicate with anyone during the exam.

☐ I am screen recording, sharing my entire desktop, unmuted and have quit and closed all tabs and background applications other than this PraireLearn tab, the Exams Ed, Zoom or my local recording software, and the Exam Proctoring Policy.

☐ I understand that I should periodically check the Exam Clarification post on the Exams Ed for important clarifications about the exam. This is my responsibility and failure to do so could result in me not getting all information about the exam.

☐ I understand that the 2 extra minutes I am given are only to **double check** I saved my answers. I understand that any unsaved answers will not be graded, and I should click the "save" button often and always before moving onto a new question.

☐ I understand that if I alter the skeleton code (unless otherwise specified), go over the line limit, or fail to follow any restrictions given in the problem statement of a coding question, then my answer will not be graded.

2 Mechanical Sorting

a) (1.5 Points). Suppose that we have the array below.

4	2	1	7	3	6	0

For each subpart, assume we are working with the original state of the array. For each algorithm, assume it is implemented as explained in lab. By a swap, we mean the exchange of two elements within the same array:

```
void swap(int[] x, int a, int b) {
    int temp = x[a];
    x[a] = x[b];
    x[b] = temp;
}
```

Note that if indices `a` and `b` are equal, i.e. if we try to swap an element with itself, it does **not** count as a swap.

i) (0.5 Points). Run selection sort. After how many swaps does 6 *first* go in front of 7?

○ 1 ○ 2 ○ 3 [✓ ○ 4] ○ 5 ○ 6 ○ 7 ○ 8 ○ 9 ○ 10
○ 11

ii) (0.5 Points). Run insertion sort. After how many swaps does 6 *first* go in front of 7?

○ 1 ○ 2 ○ 3 ○ 4 ○ 5 [✓ ○ 6] ○ 7 ○ 8 ○ 9 ○ 10
○ 11

iii) (0.5 Points). Run heapsort. Assume it is implemented in-place with bottom up heapification. Swaps during the bottom up heapification process should be counted. After how many swaps does 6 *first* go in front of 7?

[✓ ○ 1] ○ 2 ○ 3 ○ 4 ○ 5 ○ 6 ○ 7 ○ 8 ○ 9 ○ 10
○ 11

b) (0.5 Points). Suppose we have the array below.

109	123	901	442	244	321	551	155

Run LSD radix sort. What is the state of the array in the last pass *before* it is fully sorted? Format your answer as a space separated list. For instance, if the array is {`1, 2, 3`}, it would be formatted as `1 2 3`.

State: 901 109 321 123 442 244 551 155

c) (1.5 Points). Suppose we have the class `Olympian` below:

Suppose we want to sort all the `Olympians` in ascending order by their `medalCount` breaking ties alphabetically by `event` and then `name`. For simplicity, assume `names` are unique. For instance, if we have the Olympians below:

```
Olympian("Sean", "Fencing", 0)
Olympian("Jay", "Basketball", 3)
Olympian("Sohum", "Speedwalking", 0)
Olympian("Aniruth", "Basketball", 3)
Olympian("Sofia", "Speedwalking", 3)
Olympian("Zoe", "Speedwalking", 3)
Olympian("Allyson", "Speedwalking", 3)
```

we would sort them as:

```
Olympian("Sean", "Fencing", 0)
Olympian("Sohum", "Speedwalking", 0)
Olympian("Aniruth", "Basketball", 3)
Olympian("Jay", "Basketball", 3)
Olympian("Allyson", "Speedwalking", 3)
Olympian("Sofia", "Speedwalking", 3)
Olympian("Zoe", "Speedwalking", 3)
```

To achieve this, you may call each of `insertionSort`, `selectionSort`, and `mergeSort` exactly once. `sortingKey` should be one of "name", "event", or "medalCount". The signatures of the methods are below:

```
public List<Olympian> insertionSort(List<Olympian> olympians, String sortingKey)
public List<Olympian> selectionSort(List<Olympian> olympians, String sortingKey)
public List<Olympian> mergeSort(List<Olympian> olympians, String sortingKey)
```

```
public List<Olympian> sortOlympians(List<Olympian> olympians) {
    olympians = selectionSort(olympians, "name");
    olympians = mergeSort(olympians, "event");
    olympians = insertionSort(olympians, "medalCount");
    return olympians;
}
```

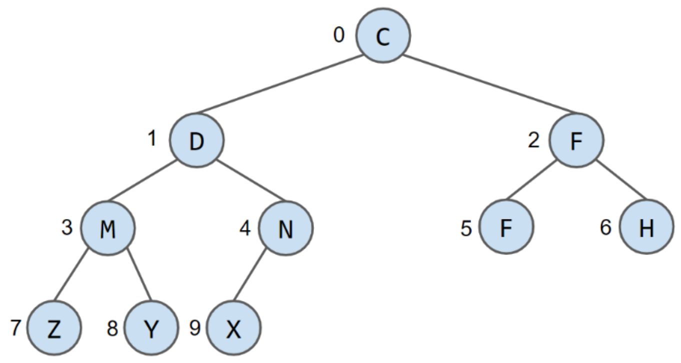
3 Representing Heaps the Hard Way

In lecture, we implemented a heap based priority queue using an array. We could also have done so with the tree representation, given below:

```
private class HeapNode {
    private T item;
    private HeapNode left;
    private HeapNode right;

    HeapNode(T i) {
        item = i;
    }
}
```

One issue with this approach is that it is somewhat tricky to find where to insert new nodes. In this problem, we'll walk through this in some detail.

a) (0.25 Points). Suppose we have a heap that currently looks like the figure below, where the integer beside each node is the "node number", i.e. position in the heap, of each item.



If we add a new item, which node will be the parent of the new node (before doing any bubbleUp operations)?

○ C ○ D ○ F ○ M [✓ N] ○ F ○ H ○ Z ○ Y ○ X

b) (1 Point). Fill in the `isRightChild(int i)` and `parent(int i)` methods. Assume for `isRightChild` that **i is nonnegative** and for `parent` that **i is positive**. Notice that **the root has node number zero.**

- `isRightChild` returns true if the node number `i` corresponds to a right child
  - `isRightChild(2)` and `isRightChild(4)` should evaluate to `true`
  - `isRightChild(0)` and `isRightChild(1)` should evaluate to `false`
- `parent` returns the node number of the parent of the given node `i`, e.g. `parent(9)` should evaluate to `4`.

```
private boolean isRightChild(int i) {
    return i % 2 == 0 && i > 0
}
```

```

private int parent(int i) {
    return _(i+ 1) / 2 -1__;
}
```

c) (1.75 Points). Fill in the `followPath(List<Boolean> path)` method so that it matches the given javadoc comment. Assume the given `path` is valid.

```
/** Return the HeapNode at the end of the given path.
 *  For example if path is: false, true, false
 *  We will go left, right, left, landing at node #9
 */
private HeapNode followPath(List<Boolean> path) {
    HeapNode p = root;
    for (_boolean boo__ : _path________) {
        if (_boo__________) {
            ________p = p.right_____________;
        } else {
            p = p.left_____________________;
        }
    }
    return p;
}
```

d) (3 Points). The method below will return the `i`th node. For example, if we call `getNode(9)` on the example heap, we will get the `HeapNode` containing **X**.

```
private HeapNode getNodeI(int i) {
    List<Boolean> pathToI = constructPath(i);
    HeapNode nodeNumberI = followPath(pathToI);
    return nodeNumberI;
}
```

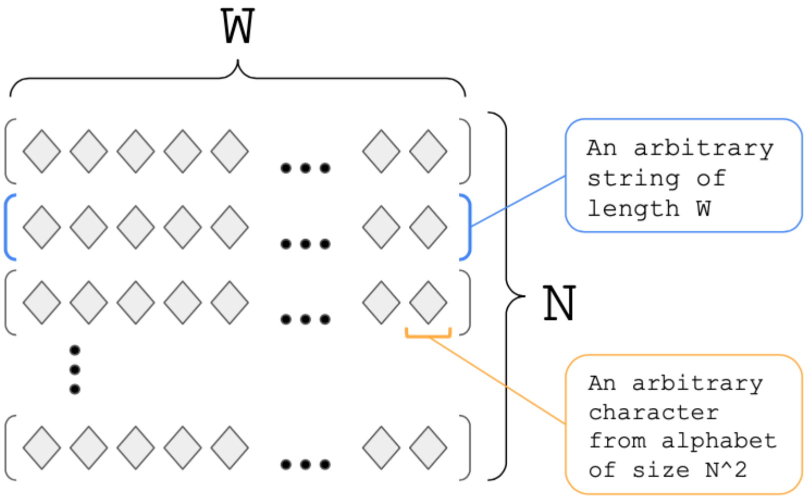
Fill in the `constructPath` method so that the `getNodeI` method above works correctly. You may use any method from the previous parts of this problem.

Hint: Recall that `LinkedList` implements the `Deque` interface, which is available on your reference sheet.

```
private List<Boolean> constructPath(int i) {
    LinkedList<Boolean> list = new LinkedList<>();
    while (_i > 0_______________________) {
        _list.addFirst(isRightChild(i))_____;
        __i = parent(i)____________________;
    }
    return list;
}
```

4 Aliens Arrive at Moon Base

a) (3 Points). Suppose the aliens have received a landing code as a series of strings, but it's all scrambled up! Help them sort the strings so they can land on the moon base! More specifically, suppose we want to **sort** N **distinct** Strings, each of length W, that are comprised of characters from an alien alphabet of size N^2. We can visualize the **set of strings** to sort as the image below.


W
N
An arbitrary string of length W
An arbitrary character from alphabet of size N^2

What are the runtimes of the following sorting algorithms, in the best and worst case? If we use some sorting algorithm, say quicksort, as the subroutine in LSD/MSD sort, that means we use quicksort to sort each *character*, exactly how counting sort is used as the subroutine. If two Strings are distinct, that means that **at least** one character differs. Finally, if the sorting algorithm provided **would not work, select "Would Not Work" for the best and worst case**. Your answers may involve both W and N.

Hint: When we compare two strings, it takes constant time in the best case, and W time in the worst case.

i) (0.75 Points). Merge sort

Best Case:

◯ $\Theta(1)$ ◯ $\Theta(logN)$ ◯ $\Theta(N)$ [✓ $\Theta(NlogN)$] ◯ $\Theta(N^2)$ ◯ $\Theta(N^2)logN)$
◯ $\Theta(logW)$ ◯ $\Theta(W)$ ◯ $\Theta(WlogW)$ ◯ $\Theta(W^2)$ ◯ $\Theta(W^2logW)$
◯ $\Theta(WlogN)$ ◯ $\Theta(WN)$ ◯ $\Theta(WNlogN)$ ◯ $\Theta(WN^2)$ ◯ $\Theta(WN^2logN)$
◯ $\Theta(NlogW)$ ◯ $\Theta(NWlogW)$ ◯ $\Theta(NW^2)$ ◯ $\Theta(NW^2logW)$
◯ $\Theta(N^2W^2)$ ◯ Never terminates (infinite loop) ◯ None of the above
Would Not Work

Worst Case:

◯ $\Theta(1)$ ◯ $\Theta(logN)$ ◯ $\Theta(N)$ ◯ $\Theta(NlogN)$ ◯ $\Theta(N^2)$ ◯ $\Theta(N^2)logN)$
◯ $\Theta(logW)$ ◯ $\Theta(W)$ ◯ $\Theta(WlogW)$ ◯ $\Theta(W^2)$ ◯ $\Theta(W^2logW)$
◯ $\Theta(WlogN)$ ◯ $\Theta(WN)$ [✓ $\Theta(WNlogN)$] ◯ $\Theta(WN^2)$ ◯ $\Theta(WN^2logN)$

○ $\Theta(NlogW)$ ○ $\Theta(NWlogW)$ ○ $\Theta(NW^2)$ ○ $\Theta(NW^2logW)$
○ $\Theta(N^2W^2)$ ○ Never terminates (infinite loop) ○ None of the above
Would Not Work

ii) (0.75 Points). Insertion sort

Best Case:
○ $\Theta(1)$ ○ $\Theta(logN)$ [✓ $\Theta(N)$] ○ $\Theta(NlogN)$ ○ $\Theta(N^2)$ ○ $\Theta(N^2)logN)$
○ $\Theta(logW)$ ○ $\Theta(W)$ ○ $\Theta(WlogW)$ ○ $\Theta(W^2)$ ○ $\Theta(W^2logW)$
○ $\Theta(WlogN)$ ○ $\Theta(WN)$ ○ $\Theta(WNlogN)$ ○ $\Theta(WN^2)$ ○ $\Theta(WN^2logN)$
○ $\Theta(NlogW)$ ○ $\Theta(NWlogW)$ ○ $\Theta(NW^2)$ ○ $\Theta(NW^2logW)$
○ $\Theta(N^2W^2)$ ○ Never terminates (infinite loop) ○ None of the above
Would Not Work

Worst Case:
○ $\Theta(1)$ ○ $\Theta(logN)$ ○ $\Theta(N)$ ○ $\Theta(NlogN)$ ○ $\Theta(N^2)$ ○ $\Theta(N^2)logN)$
○ $\Theta(logW)$ ○ $\Theta(W)$ ○ $\Theta(WlogW)$ ○ $\Theta(W^2)$ ○ $\Theta(W^2logW)$
○ $\Theta(WlogN)$ ○ $\Theta(WN)$ ○ $\Theta(WNlogN)$ [✓ $\Theta(WN^2)$] ○ $\Theta(WN^2logN)$
○ $\Theta(NlogW)$ ○ $\Theta(NWlogW)$ ○ $\Theta(NW^2)$ ○ $\Theta(NW^2logW)$
○ $\Theta(N^2W^2)$ ○ Never terminates (infinite loop) ○ None of the above
Would Not Work

iii) (0.75 Points). LSD sort with counting sort as the subroutine. Recall that LSD sort conventionally uses counting sort as the subroutine. When we run counting sort, assume that we translate each character into a number between 1 and N^2.

Best Case:
○ $\Theta(1)$ ○ $\Theta(logN)$ ○ $\Theta(N)$ ○ $\Theta(NlogN)$ ○ $\Theta(N^2)$ ○ $\Theta(N^2)logN)$
○ $\Theta(logW)$ ○ $\Theta(W)$ ○ $\Theta(WlogW)$ ○ $\Theta(W^2)$ ○ $\Theta(W^2logW)$
○ $\Theta(WlogN)$ ○ $\Theta(WN)$ ○ $\Theta(WNlogN)$ [✓ $\Theta(WN^2)$] ○ $\Theta(WN^2logN)$
○ $\Theta(NlogW)$ ○ $\Theta(NWlogW)$ ○ $\Theta(NW^2)$ ○ $\Theta(NW^2logW)$
○ $\Theta(N^2W^2)$ ○ Never terminates (infinite loop) ○ None of the above
Would Not Work

Worst Case:
○ $\Theta(1)$ ○ $\Theta(logN)$ ○ $\Theta(N)$ ○ $\Theta(NlogN)$ ○ $\Theta(N^2)$ ○ $\Theta(N^2)logN)$
○ $\Theta(logW)$ ○ $\Theta(W)$ ○ $\Theta(WlogW)$ ○ $\Theta(W^2)$ ○ $\Theta(W^2logW)$
○ $\Theta(WlogN)$ ○ $\Theta(WN)$ ○ $\Theta(WNlogN)$ [✓ $\Theta(WN^2)$] ○ $\Theta(WN^2logN)$
○ $\Theta(NlogW)$ ○ $\Theta(NWlogW)$ ○ $\Theta(NW^2)$ ○ $\Theta(NW^2logW)$
○ $\Theta(N^2W^2)$ ○ Never terminates (infinite loop) ○ None of the above
Would Not Work

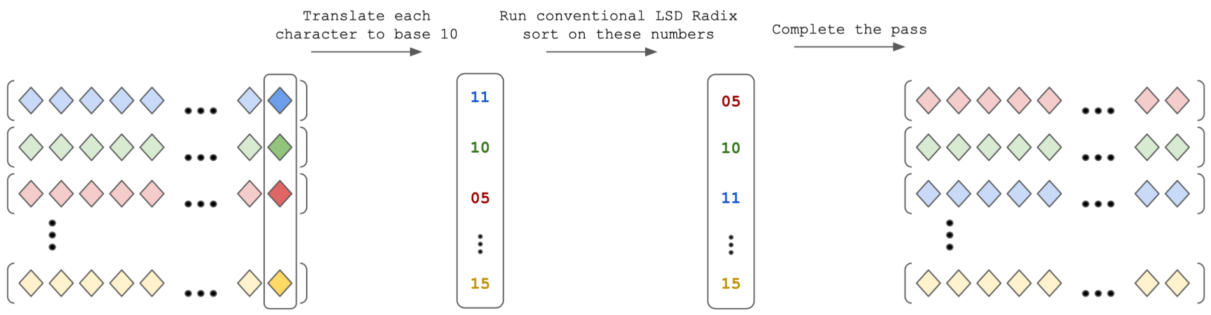
iv) (0.75 Points). MSD sort with heap sort as the subroutine.

Best Case:
○ $\Theta(1)$ ○ $\Theta(logN)$ ○ $\Theta(N)$ [✓ $\Theta(NlogN)$] ○ $\Theta(N^2)$ ○ $\Theta(N^2)logN)$
○ $\Theta(logW)$ ○ $\Theta(W)$ ○ $\Theta(WlogW)$ ○ $\Theta(W^2)$ ○ $\Theta(W^2logW)$

○ $\Theta(WlogN)$ ○ $\Theta(WN)$ ○ $\Theta(WNlogN)$ ○ $\Theta(WN^2)$ ○ $\Theta(WN^2logN)$
○ $\Theta(NlogW)$ ○ $\Theta(NWlogW)$ ○ $\Theta(NW^2)$ ○ $\Theta(NW^2logW)$
○ $\Theta(N^2W^2)$ ○ Never terminates (infinite loop) ○ None of the above
Would Not Work

Worst Case:
○ $\Theta(1)$ ○ $\Theta(logN)$ ○ $\Theta(N)$ ○ $\Theta(NlogN)$ ○ $\Theta(N^2)$ ○ $\Theta(N^2)logN)$
○ $\Theta(logW)$ ○ $\Theta(W)$ ○ $\Theta(WlogW)$ ○ $\Theta(W^2)$ ○ $\Theta(W^2logW)$
○ $\Theta(WlogN)$ ○ $\Theta(WN)$ [✓ $\Theta(WNlogN)$] ○ $\Theta(WN^2)$ ○ $\Theta(WN^2logN)$
○ $\Theta(NlogW)$ ○ $\Theta(NWlogW)$ ○ $\Theta(NW^2)$ ○ $\Theta(NW^2logW)$
○ $\Theta(N^2W^2)$ ○ Never terminates (infinite loop) ○ None of the above
Would Not Work

b) (2 Points). For the following two parts, we will be calling **LSD sort with conventional LSD sort as the subroutine**! Conventional LSD sort uses counting sort as the subroutine. Since conventional LSD sort requires that the objects it's comparing are comprised of **digits of a fixed radix, or base**, we will decompose each character from the alien alphabet into a number of a certain radix. We can visualize the first pass of the sorting algorithm as shown in the image below.


Translate each character to base 10
Run conventional LSD Radix sort on these numbers
Complete the pass
11
10
05
15
05
10
11
15

For each part below, find *only* the **worst case** runtime of calling **LSD sort with LSD sort as the subroutine**, if we decompose each character into a number of the given radix.

Here are some helpful hints before you begin.

- If we decompose a character of size M into a number of radix B, the number of digits in the resulting number is $log_B M$.
- $log_B B = 1$.
- $log_B M^a = a \times log_B M$.

i) (1 Point). Radix 10.

○ $\Theta(1)$ ○ $\Theta(logN)$ ○ $\Theta(N)$ ○ $\Theta(NlogN)$ ○ $\Theta(N^2)$ ○ $\Theta(N^2)logN)$
○ $\Theta(logW)$ ○ $\Theta(W)$ ○ $\Theta(WlogW)$ ○ $\Theta(W^2)$ ○ $\Theta(W^2logW)$
○ $\Theta(WlogN)$ ○ $\Theta(WN)$ [✓ $\Theta(WNlogN)$] ○ $\Theta(WN^2)$ ○ $\Theta(WN^2logN)$
○ $\Theta(NlogW)$ ○ $\Theta(NWlogW)$ ○ $\Theta(NW^2)$ ○ $\Theta(NW^2logW)$

◯ $\Theta(N^2W^2)$ ◯ Never terminates (infinite loop) ◯ None of the above
Would Not Work

ii) (1 Point). Radix N.

◯ $\Theta(1)$ ◯ $\Theta(logN)$ ◯ $\Theta(N)$ ◯ $\Theta(NlogN)$ ◯ $\Theta(N^2)$ ◯ $\Theta(N^2)logN)$
◯ $\Theta(logW)$ ◯ $\Theta(W)$ ◯ $\Theta(WlogW)$ ◯ $\Theta(W^2)$ ◯ $\Theta(W^2logW)$
◯ $\Theta(WlogN)$ ◯ $\Theta(WN)$ ◯ $\Theta(WNlogN)$ [✓ $\Theta(WN^2)$] ◯ $\Theta(WN^2logN)$
◯ $\Theta(NlogW)$ ◯ $\Theta(NWlogW)$ ◯ $\Theta(NW^2)$ ◯ $\Theta(NW^2logW)$
◯ $\Theta(N^2W^2)$ ◯ Never terminates (infinite loop) ◯ None of the above
Would Not Work

5 Hashing

a) (2 Points). Suppose we have the Dog class below.

```java
public class Dog {
    private static int dogsCreated = 0;
    private int age;
    private final int microchipID;
    public String name;
    public String owner;

    public Dog(int age, String name, String owner) {
        Dog.dogsCreated += 1;
        this.age = age;
        this.name = name;
        this.owner = owner;
        this.microchipID = dogsCreated;
    }

    public void birthday() {
        this.age += 1;
    }

    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Dog)) {
            return false;
        }
        Dog other = (Dog) o;
        return this.name.equals(other.name) && this.microchipID == other.microchipID;
    }

    @Override
    public int hashCode() {
        // part a
    }
    ...
}
```

For each of the following implementations of the Dog's `hashCode` method, determine if it is good and valid, bad but valid, or invalid.

i) (0.5 Points)

```java
@Override
public int hashCode() {
    return this.age;
}
```

○ good and valid ○ bad but valid [redacted] invalid

ii) (0.5 Points)

```
@Override
public int hashCode() {
    return this.microchipID;
}
```

[✓ good and valid] ◯ bad but valid ◯ invalid

iii) (0.5 Points)

```
@Override
public int hashCode() {
    return (int) this.name.charAt(0);
}
```

◯ good and valid [✓ bad but valid] ◯ invalid

iv) (0.5 Points)

```
@Override
public int hashCode() {
    return this.owner.hashCode();
}
```

[✓ good and valid] ◯ bad but valid ◯ invalid

b) (0.5 Points). If a `HashMap` contains two or more items with equal `keys`, how might have this happened? Give one reason. Keep your answer to 2 sentences. If you write more than two sentences, we will only consider the first two. If you give multiple reasons, we will only consider the first. Assume that we are referring to the built in java `HashMap`. Note that this part is independent from the previous, i.e. the `keys` aren't necessarily `Dogs`.

`Reason:`

c) (3.5 Points). Implement the method `isUnique` in the `HashMap` class that returns `true` if and only if the `HashMap` has **unique** keys, i.e. it does *not* contain two or more equal `keys`. Assume all implementations of the `List` and `Set` interface have been imported.

Hint: Consider objects with *invalid* hashCodes.

```
public class 61BLHashMap<K, V> implements Map61BL<K, V> {
    LinkedList<Entry<K, V>>[] map;
    int size;

    public boolean isUnique() {
        List<K> visited = new List<k>()____________;
        for (LinkedList l: map_____________________) {
            for (Entry x: l________________________) {
                if (visited.contains(x)____________) {
                    return false;__________________;
                }
                visited.add(x)_____________________;
            }
        }
        return true;_______________________________;
    }

    ...

    private static class Entry<K, V> {
        private K key;
        private V value;

        Entry(K key, V value) {
            this.key = key;
            this.value = value;
        }
    }
}
```

6 Heaps

a) (2.5 Points).

i) (1 Point). Suppose we have the min-heap below (represented as an array) with **distinct** elements, where the values of A and B are unknown. Note that A and B aren't necessarily integers.

`{1, A, 3, 5, 9, 11, 13, 10, B}`

What can we say about the relationships between the following elements? Put >, <, or ? if the answer is not known.

A [✓ ◯ >] ◯ < ◯ ? 1

A ◯ > ◯ < [✓ ◯ ?] 3

B ◯ > ◯ < [✓ ◯ ?] 10

A ◯ > [✓ ◯ <] ◯ ? B

ii) (1.5 Points). Note for both parts below, the values of `A` and `B` should **not** violate the min-heap properties. Put `-inf` or `inf` if there isn't a lower or upper bound, respectively. If the bound for `B` depends on the value of `A`, or vice versa, you may put the variable in the bound, e.g. `A < B`.

Considering **one `removeMin`** call, put **tight** bounds on `A` and `B` such that:

- We perform the **maximum** number of swaps.

  1 < **A** < 3

  10 < **B** < inf

- We perform the **minimum** number of swaps.

  3 < **A** < 5

  5 < **B** < 11

b) (1 Points). Run **bottom up heapification** on the array below to create a ***min* heap**. Recall that when we run bottom up heapification, we call `bubbleDown` on the elements of the array in reverse order.

`{6, 5, 4, 3, 2, 1, 0}`

i) (0.5 Points). How many swaps were performed?

◯ 1 ◯ 2 ◯ 3 [✓ 4] ◯ 5 ◯ 6 ◯ 7 ◯ 8 ◯ 9 ◯ 10

ii) (0.5 Points). What is the final state of the array?

○ `{0, 1, 2, 3, 5, 6, 4}` ○ `{0, 2, 1, 3, 4, 5, 6}` [✓ `{0, 2, 1, 3, 5, 6, 4}`] ○ `{0, 1, 2, 3, 4, 5, 6}` ○ None of the above

c) (2 Points). Using the array below, `{6, W, 5, 3, X, Y, Z}` assign, in *some* order, the variables **W**, **X**, **Y**, and **Z** to values `0, 1, 2, 4`, such that when we run bottom up heapification to create a **min-heap**, the array is **fully sorted**:

```
{0, 1, 2, 3, 4, 5, 6}
```

The bottom up heapification process should have **as many swaps as possible.** That is, if there are multiple viable arrays that produce the fully sorted array, put the one that has the most swaps.

W: ○ 0 ○ 1 ○ 2 [✓ 4]
X: ○ 0 [✓ 1] ○ 2 ○ 4
Y: [✓ 0] ○ 1 ○ 2 ○ 4
Z: ○ 0 ○ 1 [✓ 2] ○ 4

7 DisjointSetsGraph

(4.25 Points). Recall the `DisjointSets` interface.

```java
public interface DisjointSets {
    /** connects two items P and Q */
    void connect(int p, int q);

    /** checks to see if two items are connected */
    boolean isConnected(int p, int q);
}
```

The `DisjointSets` implementation you all are most familiar with is the `WeightedQuickUnion`, but there are many other possible implementations! In this problem, we will implement the `DisjointSetsGraph` class that uses a graph to store the connected components. More specifically, the `DisjointSetsGraph` class will store, as an instance attribute, an adjacency list represented implementation of the undirected `Graph` interface provided below:

```java
public interface Graph {

    /* Adds the undirected edge v-w to this graph. If the edge exists, it does nothing. */
    void addEdge(int v, int w);

    /* Returns the vertices adjacent to vertex v. */
    Iterable&lt;Integer> adj(int v);

    /* Returns the number of vertices in the graph.
       Assume the vertices in the graph are numbered 0 to size() - 1 */
    int size();
}
```

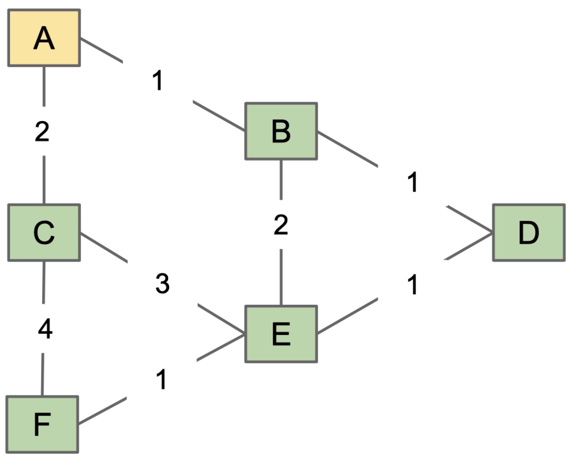
Implement the `connect` and `isConnected` methods below so that the **runtime of each is O(V)** where V is the number of vertices in the Graph.

If you don't meet the runtime bounds, i.e. the runtime of either `isConnected` or `connect` is worse than **O(V)**, you will only be eligible for partial credit. Assume all implementations of the `List`, `Set`, and `Map` interface have been imported.

```java
import java.util.*;

public class DisjointSetsGraph implements DisjointSets {
    Graph graph;

    @Override
    public void connect(int p, int q) {
        /* You may add a maximum of 3 lines of code */
          if (! isConnecte(p, q)) {
             graph.addEdge(p, q)
          }

    }

    @Override
    public boolean isConnected(int p, int q) {
        List<Integer> fringe = new LinkedList<>();
        fringe.add(p);
        ______________________________________________;
        while (!fringe.isEmpty()) {
            /* You may add at most 10 lines of code in this while loop */
            int k = fringe.remove(0);
            if  (k == q) {
               return true;
            }
            for (int i: graph.adj(k)) {
               if (!visited.contains(i)) {
                  visited.add(i);
                  fringe.add(i);
               }
            }


        }
        return false
               ________________________________________;
    }
}
```

8 Graph Algorithms

a) (4 Points). Suppose we have the graph below.



For all parts, break ties alphabetically. We recommend drawing this graph on paper! Recall that SPT stands for shortest paths tree.

i) (1 Points). Run Dijkstra's from vertex `A` on the graph above.

1. **(0.5 Points).** What is the ordering that vertices are visited? Format your answer as a space separated list, e.g. `A B C D E F`.

   Order: A B C D E F

2. **(0.5 Points).** What edges are included in the SPT rooted at A?

   [✓ ☐ AB] [✓ ☐ AC] [✓ ☐ BD] [✓ ☐ BE] ☐ CE ☐ CF ☐ DE [✓ ☐ EF]

ii) (1 Points). Run A* from vertex `A` to `F` on the graph above. Let the heuristic of a vertex `v` be the number of edges between `v` and `F` in the BFS tree rooted at `F` (you may need to run BFS to find this). For instance, the heuristic of D is 2.

1. **(0.5 Points).** What is the ordering that vertices are visited? Format your answer as a space separated list, e.g. `A B C D E F`.

   Order: A B C D E F

2. **(0.5 Points).** What edges comprise the shortest path?

   [✓ ☐ AB] [✓ ☐ AC] [✓ ☐ BD] [✓ ☐ BE] ☐ CE ☐ CF ☐ DE [✓ ☐ EF]

iii) (1 Points). Run Prim's from vertex `A` on the graph above.

1. **(0.5 Points).** What is the ordering that vertices are visited? Format your answer as a space separated list, e.g. `A B C D E F`.

   Order: A B D E F C

2. **(0.5 Points).** What edges are included in the MST rooted at A?

   [✓ ☐ AB] [✓ ☐ AC] [✓ ☐ BD] ☐ BE ☐ CE ☐ CF [✓ ☐ DE] ☐ EF

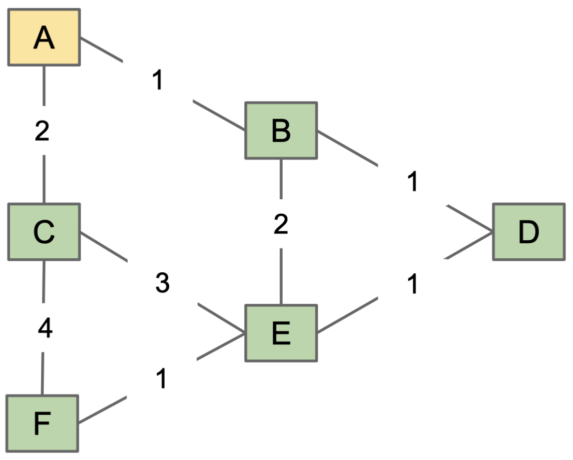
iv) (0.5 Points). Increase the weight of one edge by 1 so that the SPT found by Dijkstra's is the same as the MST found by Prim's. Assume we've run Prim's and Dijkstra's from vertex `A` on the graph above.

If the SPT found by Dijkstra's is already the same as the MST found by Prim's, select the option "no change needed". If it's impossible for them to be equal after increasing the weight of one edge by 1, select the option "impossible".

◯ AB ◯ AC ◯ BD [✓ ◯ BE] ◯ CE ◯ CF ◯ DE ◯ EF
◯ no change needed ◯ impossible

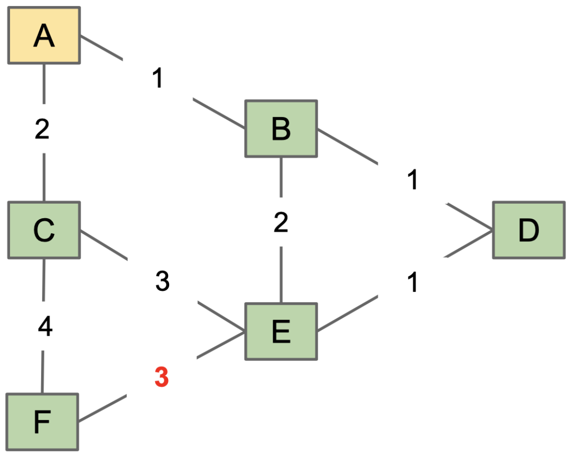
b) (2 Points). Multiple SPTs

i) (0.5 Points). Using the graph from the previous part, which has been recopied for your convenience below, how many different **SPTs** rooted at A exist? Recall there can be multiple SPTs for a given start vertex if there are multiple paths of equal, minimum cost to at least one end vertex.



Number of SPTs: ◯ 0 ◯ 1 [✓ ◯ 2] ◯ 3 ◯ 4 ◯ 5 ◯ 6 ◯ 7
◯ 8

ii) (0.5 Points). Now, suppose we change the edge weight of `EF` to 3. How many SPTs are there now? The altered graph has been drawn for your convenience below. Hint: The number of SPTs should *increase* from part i to part ii.



Number of SPTs: ◯ 0 ◯ 1 ◯ 2 ◯ 3 [✓ 4] ◯ 5 ◯ 6 ◯ 7 ◯ 8

iii) (1 Point). Finally, using the same graph from part b.ii, change *one* edge weight to further **increase** the total number of SPTs rooted at A.

Edge: ◯ AB ◯ AC ◯ BD ◯ BE [✓ CE] ◯ CF ◯ DE ◯ EF

New weight: ◯ 0 [✓ 1] ◯ 2 ◯ 3 ◯ 4 ◯ 5 ◯ 6 ◯ 7 ◯ 8

9 Hopping Iterator

Hopping Iterator (5 points). A `HoppingIterator` accepts an int array of numbers as input to its constructor, and iterates through the array as follows:

- It always begins at index zero. That is, the first value it returns will be `input[0]`.
- By default, the next index it will visit will be its previous return value. However, if that next index is either invalid or that index has already been visited, it instead visits the earliest unvisited index.

For instance, if we passed in the array {`5, 2, -11, 9, 4, 1, 0, 11`}, then we would:

1. Begin at index 0 and return 5.
2. Go to index 5 and return 1.
3. Go to index 1 and return 2.
4. Go to index 2 and return -11.
5. Try going to index -11, but this is an invalid index, so we restart at the earliest unvisited spot, which is index 3 and return 9.
6. Try going to index 9, but this is an invalid index, so we restart at the earliest unvisited spot, which is index 4 and return 4.
7. Try going to index 4, but 4 has already been visited, so we restart at the earliest unvisited spot, which is index 6 and return 0.
8. Try going to index 0, but 0 has already been visited, so we restart at the earliest unvisited spot, which is index 7 and return 11.

Now, we have finished iterating, so we will not return any more values. As a specific example, if we run the code on the example above, we'll get the output `5 1 2 -11 9 4 0 11`.

a) (0.25 Points). What will be the output of a `HoppingIterator` on the input {`4, 1, 9, 6, -5, 0, 11`}? Please separate the numbers with **spaces**, matching the format for the output above.

Output: 4, -5, 1, 9, 6, 11, 0

b) (4.75 Points). Fill in the implementation for `HoppingIterator` below. You may assume the implementations of the `Set`, `List`, and `Map` interfaces shown on the reference sheet have been imported.

```
public class HoppingIterator implements Iterator<Integer> {
    private int[] arr;
    // You may add at most 3 instance variables
     private int index;
    public HoppingIterator(int[] input) {
        arr = input;
        // You may add at most 3 lines below (hint: instantiate your other instance variables)
        index = 0;
    }

    @Override
    public boolean hasNext() {
        return index < arr.length;
    }

    @Override
    public Integer next() {
        int ans = _________________________;
        _________________________________;
        if (______________________________) {
            _____________________________;
        } else {
            while (_______________________) {
                _________________________;
            }
            _____________________________;
        }
        return ans;
    }
}
```

10 MST Adaptations

For each subpart below, determine whether the algorithm presented on the described graph will always, sometimes, or never find a valid MST. Assume that the original graph is

- simple
- connected
- undirected
- has **at least one cycle**
- and **the weights of all the edges are distinct**.

If you skimmed the assumptions above, read them again!

Assume ties are broken randomly in each algorithm presented. Recall that an MST is simply a set of edges. Parts are independent.

a) (0.5 Points). For this part, additionally assume the original graph has at least one negative edge. Run Kruskal's as normal.

[✓ ●] Always finds a valid MST.
○ Sometimes finds a valid MST.
○ Never finds a valid MST.

b) (0.75 Points). For this part, additionally assume the graph has **exactly one cycle**. Run DFS from a random vertex breaking ties by edge weight. Output the DFS tree.

○ Always finds a valid MST.
[✓ ●] Sometimes finds a valid MST.
○ Never finds a valid MST.

c) (0.75 Points). Run Dijkstra's from every vertex in the graph for **one step** so that from each vertex we have found *one* edge. Add these edges to a set. Output the set.

○ Always finds a valid MST.
[✓ ●] Sometimes finds a valid MST.
○ Never finds a valid MST.

Each of the remaining subparts will *modify* the original graph, e.g. adding edges or changing edge weights. Your task is to determine whether the algorithm presented on the modified graph will always, sometimes, or never find a valid MST of the **original, unmodified graph**. Assume the modifications are **in addition** to the assumptions listed in the beginning. Parts are independent.

d) (0.5 Points). Find the minimal edge **M** in the graph, and modify the original graph by adding |**M**| (the edge weight of **M**) to all edges in graph. Run Kruskal's on the modified graph.

[✓ Always finds a valid MST of the original, unmodified graph.]
◯ Sometimes finds a valid MST of the original, unmodified graph.
◯ Never finds a valid MST of the original, unmodified graph.

e) (0.75 Points). For this part, additionally assume the original graph has at least one negative edge. Modify the original graph by multiplying each **negative** edge by -1. Run Kruskal's on the modified graph.

◯ Always finds a valid MST of the original, unmodified graph.
[✓ Sometimes finds a valid MST of the original, unmodified graph.]
◯ Never finds a valid MST of the original, unmodified graph.

f) (0.75 Points). For this part, additionally assume the original graph is fully connected *and* every vertex is incident to at least one negative edge. Recall a fully connected graph means an edge exists between every pair of vertices. Modify the original graph by multiplying each **negative** edge by -1. Run Kruskal's on the modified graph.

◯ Always finds a valid MST of the original, unmodified graph.
[✓ Sometimes finds a valid MST of the original, unmodified graph.]
◯ Never finds a valid MST of the original, unmodified graph.

g) (0.75 Points). For this part, additionally assume the original graph is fully connected *and* every vertex is incident to at least one negative edge. Modify the original graph by multiplying each edge by -1. Run Kruskal's on the modified graph.

◯ Always finds a valid MST of the original, unmodified graph.
◯ Sometimes finds a valid MST of the original, unmodified graph.
[✓ Never finds a valid MST of the original, unmodified graph.]

h) (0.75 Points). For this part, additionally assume the graph isn't fully connected. Modify the graph by the following: for every pair of vertices without an edge between them, add an edge with weight equal to the weight of the **largest** edge. Run Kruskal's on the modified graph.

◯ Always finds a valid MST of the original, unmodified graph.
[✓ Sometimes finds a valid MST of the original, unmodified graph.]
◯ Never finds a valid MST of the original, unmodified graph.

i) (0.75 Points). For this part, additionally assume the graph isn't fully connected. Modify the graph by the following: for every pair of vertices without an edge between them, add an edge with weight equal to the weight of the **smallest** edge. Run Kruskal's on the modified graph.

- ○ Always finds a valid MST of the original, unmodified graph.
- ○ Sometimes finds a valid MST of the original, unmodified graph.
- [✓ Never finds a valid MST of the original, unmodified graph.]

11 Frauds List

Note: This problem is difficult and we recommend working on it after finishing the rest of the exam.

(6 Points). Suppose we have the IntList and FraudsList classes below.

```
public class IntList {
    public int first;
    public IntList rest;

    public IntList(int f, IntList r) {
        first = f;
        rest = r;
    }

    public int size() {
        IntList p = this;
        int totalSize = 0;
        while (p != null) {
            totalSize += 1;
            p = p.rest;
        }
        return totalSize;
    }
}

class FraudList extends IntList {
    public FraudList(int f, IntList r) {
        super(f, r);
    }
    public int size() {
        return -super.size();
    }
}
```

Implement the method findFrauds which accepts an array of IntLists in which some of the elements are, or may contain, FraudLists! That is, the dynamic type of certain IntList instances is FraudList. As shown above, a FraudList is an IntList whose size method returns the negative of the correct size. You must report these FraudLists by **non-destructively** returning a **new** FraudList of all the FraudList instances linked together in the order they appear in arr.

You may **not** modify the given array arr or the IntLists inside of FraudList. You may **not** use instanceOf, getClass(), isInstance() or any method not explicitly written in the classes above or imported. An instance of the problem is shown below:

```
IntList first = new IntList(1000, new IntList(1002, new FraudList(1, new FraudList(2, null))));
IntList second = new FraudList(3, null);
IntList third = new IntList(3000, null);
IntList fourth = new FraudList(4, new IntList(231, new FraudList(5, null)));
IntList[] arr = new IntList[]{first, second, third, fourth};
FraudList frauds = findFrauds(arr);
```

After executing the lines above, `frauds` should be equal to the `FraudList` with the elements `1, 2, 3, 4, 5` and **`arr`, as well as the contents within `arr`, should be unchanged**. Fill in the skeleton below. You may not delete, modify, or add to any of the provided skeleton code.

```
import static java.lang.System.arraycopy;

public static FraudList findFrauds(IntList[] arr) {
    IntList[] copy = new IntList[arr.length];
    arraycopy(arr, 0, copy, 0, arr.length);
    return helper(copy, 0);
}

public static FraudList helper(IntList[] copy, int index) {
    if (index == copy.length) {
        return null;
    } else if (copy[index] == null) {
        return helper(copy, index + 1]);
    }
    intList curr  = copy[index];
    copy[index] = curr.rest;
    if (curr.size() < 0) {
        return new Fraudlist(curr.ite, helper;(copy, index))
    } else {
        return helper(copy, index);
    }
}
```

CS 61BL Summer 2021 Reference Sheet

Note that not everything on this reference sheet may be needed.

System.arraycopy

```
// Copies length elements from src starting at srcPos to dest starting at destPos
// int[] can be replaced by an array holding any type
System.arraycopy(int[] src, int srcPos, int[] dest, int destPos, int length)
```

JUnit Methods

```
assertEquals(Object expected, Object actual)
assertEquals(int expected, int actual)
assertEquals(double expected, double actual)
assertTrue(boolean actual)
assertFalse(boolean actual)
assertNotNull(Object actual)
assertArrayEquals(Object[] expected, Object[] actual)
assertArrayEquals(int[] expected, int[] actual)
assertArrayEquals(double[] expected, double[] actual)
```

IntList

```
public class IntList {
    public int first;
    public IntList rest;

    public IntList(int f, IntList r) {
        first = f;
        rest = r;
    }

    /* of returns an IntList with the given numbers.
    this method can take any number of inputs and will return an IntList containing those inputs in
    the order given. For example:
    IntList L = IntList.of(1, 2);
    L.first; -> evaluates to 1
    L.rest.first; -> evaluates to 2 */
    public static IntList of(int... input) { ... }
}
```

Iterator, Iterable, Comparator, and Comparable

```
public interface Iterator<T> {
    boolean hasNext();
    T next();
}

public interface Iterable<T> {
    Iterator<T> iterator();
}

public interface Comparator<T> {
    int compare(T o1, T o2);
}

public interface Comparable<T> {
    int compareTo(T obj);
}
```

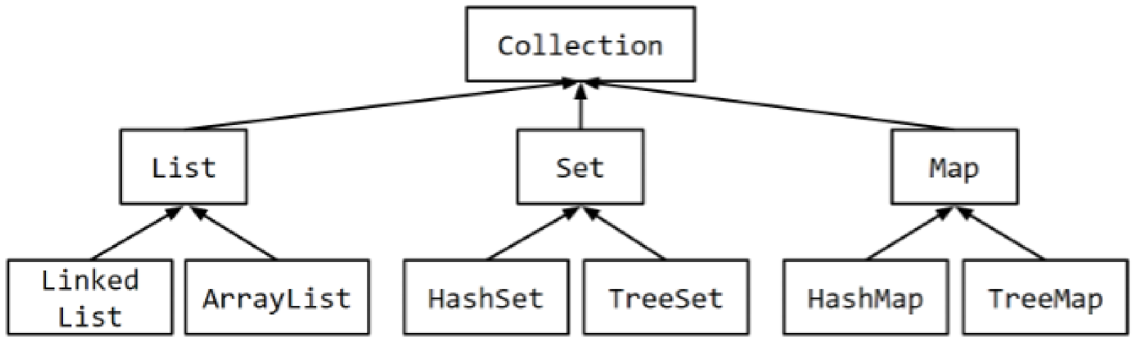
String API

```
public class String {

    /** Returns the character at the specified index (position) */
    public char charAt(int index) { ... }

    /**
     * Returns a string that is a substring of this string. The substring begins at the specified
     * beginIndex and extends to the character at index endIndex - 1. Thus the length of the
     * substring is endIndex-beginIndex.
     */
    public String substring(int beginIndex, int endIndex) { ... }

    /** Returns the length of this string. */
    public int length() { ... }
}
```

Map, Set, and List

```java
public interface Map<K, V> ... { ...
    boolean containsKey(K key);
    V get(K key);
    V getOrDefault(K key, V value);
    void put(K key, V value);
    Set<K> keySet();
    Iterator<K> iterator();
}

public interface Set<K> ... { ...
    boolean contains(K key);
    void add(K key);
    Iterator<K> iterator();
}

public interface List<T> ... { ...
    boolean contains(T item);
    void add(T item);
    void add(int index, T item);
    T get(int i);
    T set(int i, T item);
    int indexOf(Object o);
    boolean remove(Object o);
    Iterator<T> iterator();
}
```

Implementations:



More Data Structures

```
public interface Deque<T> ... { ...
    void addFirst(T item);
    void addLast(T item);
    T removeFirst();
    T removeLast();
    T getFirst();
    T getLast();
}

public class Stack<T> ... { ...
    public T pop() { ... }
    public void push(T item) { ... }
    public Iterator<T> iterator() { ... }
}

public class Queue<T> ... { ...
    public T dequeue() { ... }
    public void enqueue(T item) { ... }
    public Iterator<T> iterator() { ... }
}

public class MinPQ<T> ... { ...
    public MinPQ() { ... }
    public MinPQ(Comparator<T> c) { ... }
    public T removeSmallest() { ... }
    public T smallest() { ... }
    public void add(T item) { ... }
    public Iterator<T> iterator() { ... }
}
```